2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

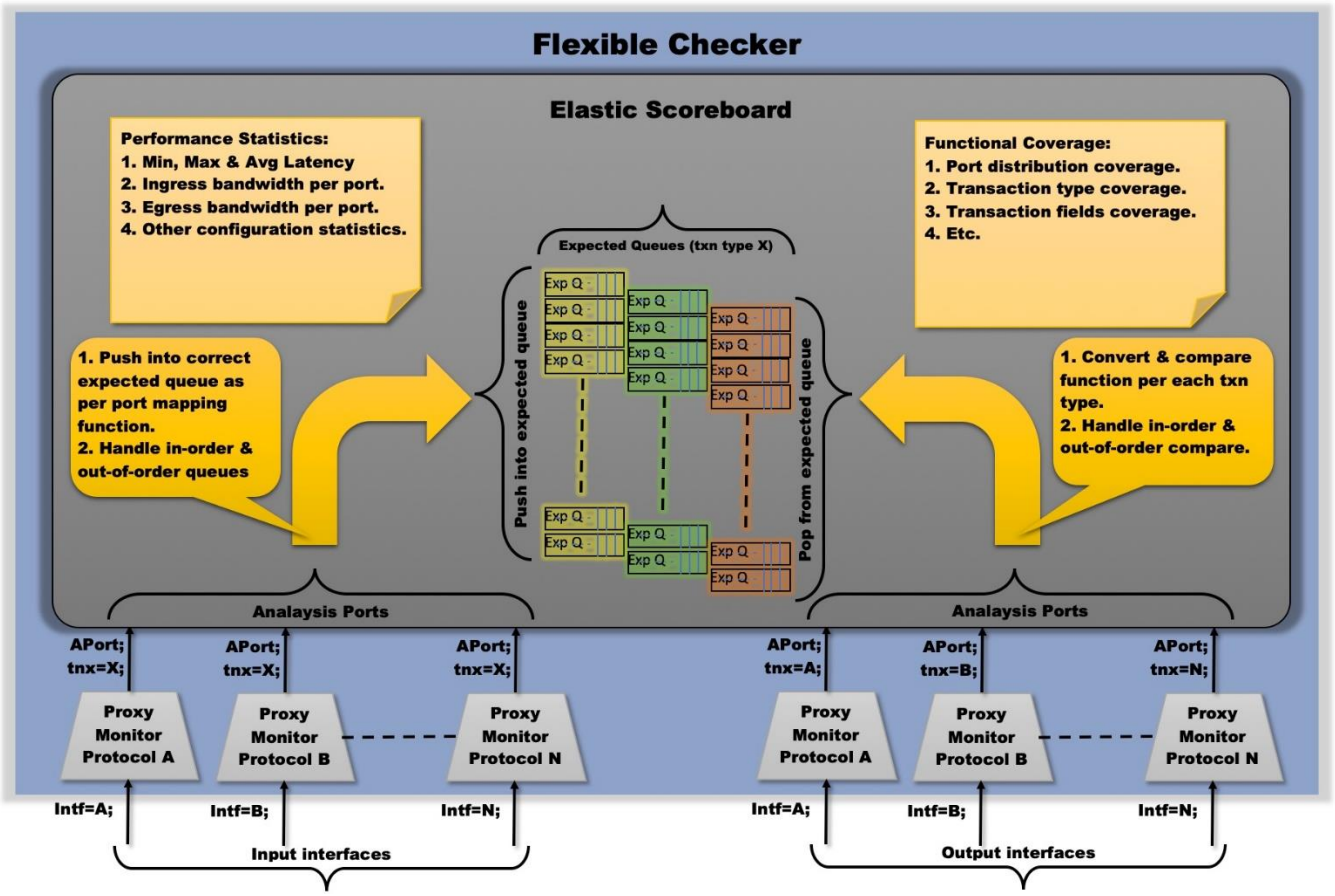
Flexible Checker: A one-stop shop for all your checkers and a methodology for elastic scoreboarding

Saad Zahid, Chandra Veedhi, Sumit Dhamanwala
Arteris Inc
9601 Amberglen Blvd. Building G, Suite 117,
Austin TX, 78729

***Abstract*—This paper aims to define a methodology for a composite checker/scoreboard called Flexible Checker that provides test-bench writers a solution that can adapt to ever-changing design configurations. Flexible Checker also provides a solution that is portable across blocks, sub-blocks, and chip-level environments. It can also support flexible "transaction trace checking," which checks transactions at multiple internal nodes before reaching their end points. Flexible Checker is equipped to record and dump vital statistics, including bandwidth, latency, traffic distribution, transaction count (type, size, etc.), from end to end in the report phase of a UVM.**

I. INTRODUCTION

The scoreboard for the UVM test bench of a SoC is a crucial piece that checks the data integrity across the DUT (design under test). The way a scoreboard is coded traditionally varied from engineer to engineer, configuration to configuration, and block to block. In this paper, we will show how to templatize a scoreboard to make it more configurable, scalable, and lightweight at the same time. To achieve this, we introduce what we call a **proxy transaction container class**, a **proxy monitor**, and **proxy transaction queues**.

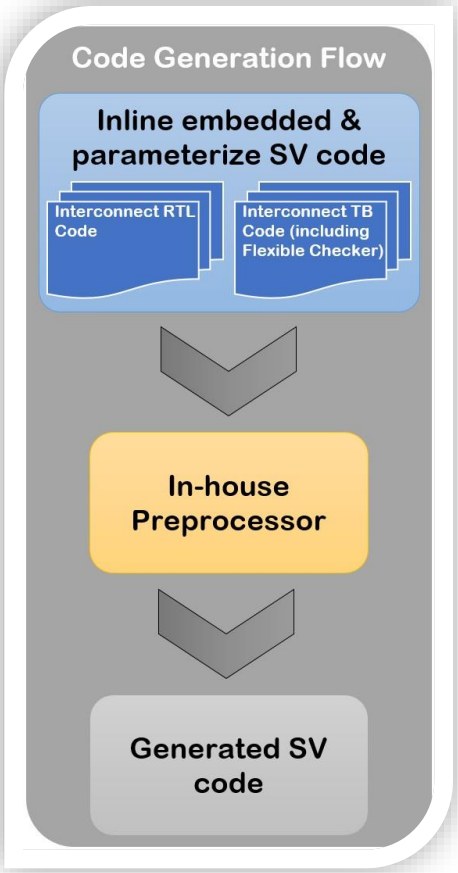


II. ARTERIS VERIFICATION CHALLENGE

For companies like Arteris, which provides IP interconnects with the capability to be constituted of an infinite number of configurations, it is essential to make the corresponding test benches fully flexible and elastic yet robust and effective to deliver high quality. Our biggest challenges to consider during the development process are as follows:

1. Each and every interface field and position is configurable.
2. Each block/unit/component is also fully elastic in terms of
    a. The number of input/outputs,
    b. how its features can be enabled/disabled from a code-generation prospective, and
    c. the buffer sizes, pipeline stages, states, registers, etc.
3. Similarly, all the above must be complemented by an equivalent test bench to verify each available feature accordingly.

These above-mentioned challenges cannot be solved just by using the SystemVerilog (SV) parameter alone, so we have to develop a custom tool to preprocess our embedded (inline) and parameterized code to generates the final SV code.



III. KEY COMPONENTS

To make Flexible Checker highly scalable, agile, and portable, we needed to make a few fundamental verification strategy decisions and develop the key components as follows:

1. All in-house UVCs (or VIPs) must have a portable monitor so that we can instantiate it directly in Flexible Checker.
2. For external (third-party) VIPs, we wrote a proxy monitor to collect third-party transactions and embed them into our proxy transaction container class and pass them to an elastic scoreboard.
3. All test-bench components must use a single transaction type (proxy transaction container) regardless of the actual transaction type (for example, AXI, APB, AHB, or any number of in-house protocols).
4. We assembled a wrapper class (extended from UVM_COMPONENT) called Flexible Checker, which includes all the required monitors and the elastic scoreboard.

IV. PROXY TRANSACTION CONTAINER CLASSES

The proxy transaction item class (3rdparty_txn_item) is both a container class for porting a third-party VIP as well as a base class for an in-house VIP. It is the transaction class from which other protocol-specific transaction items are derived. It also has a handle defined as the uvm_sequence_item type, which is used as a container for third-party VIP transaction items. It defines transaction query operations such as “get destination,” “get source,” and “get ID” as virtual functions, which are supposed to be overloaded in the extended classes. The code is shown below.

flexible_seq_item.sv

```
`ifndef __FLEXIBLE_SEQ_ITEM__
 `define __FLEXIBLE_SEQ_ITEM__

class flexible_seq_item extends uvm_sequence_item;
   uvm_sequence_item txn_handle;

   // Listing all the virtual function to be extended in derived
   // UVC sequence item
   virtual function new(string name);
      super.new(name);
   endfunction: new

   virtual function bit [4095:0] get_data();
      `uvm_fatal(get_name(), "base get_data() called")
   endfunction: get_data

   virtual function int get_addr();
      `uvm_fatal(get_name(), "base get_addr() called")
   endfunction: get_addr

   virtual function int get_source();
      `uvm_fatal(get_name(), "base get_source() called")
   endfunction: get_source

   virtual function int get_dest();
      `uvm_fatal(get_name(), "base get_dest() called")
   endfunction: get_destination

   virtual function void my_compare(flexible_seq_item txn);
      `uvm_fatal(get_name(), "base my_compare() called")
   endfunction: my_compare
endclass: flexible_seq_item

`endif // __FLEXIBLE_SEQ_ITEM__

```

3rdPartyVIP_Seq_Item.sv

```
`ifndef __3rdpartyvip_seq_item__
 `define __3rdpartyvip_seq_item__

class 3rdpartyvip_txn_item extends flexible_seq_item;

   // note txn_handle needs to be assigned to the extvip_seq_item

   // extend the virtual functions
   function new(string name);
      super.new(name);
   endfunction: new

   virtual function bit [4095:0] get_data();
      extvip_seq_item e_item;
      $cast(e_item,txn_handle);
      return serialse(e_item.data_arr);
   endfunction : get_data

   virtual function int get_id();
      extvip_seq_item e_item;
      $cast(e_item,txn_handle);
      return e_item.id;
   endfunction : get_id

   virtual function int get_addr();
      extvip_seq_item e_item;
      $cast(e_item,txn_handle);
      return e_item.addr;
   endfunction : get_addr

   virtual function int get_source();
      extvip_seq_item e_item;
      $cast(e_item,txn_handle);
      return  e_item.instance_no;
   endfunction : get_source

   virtual function int get_dest();
      extvip_seq_item e_item;
      $cast(e_item,txn_handle);
      return address_map_lookup(e_item.addr);
   endfunction : get_dest

   virtual function void my_compare(flexible_seq_item txn);
      extvip_seq_item e_item;
      $cast(e_item, txn_handle);

      if(e_item.get_data() != txn.get_data()) `uvm_error(get_name(), "Data mismatched!")
      if(e_item.get_addr() != txn.get_addr()) `uvm_error(get_name(), "Address mismatched!")
   endfunction: my_compare
endclass: 3rdpartyvip_seq_item
`endif // __3rdpartyvip_seq_item__

```

V. PROXY MONITOR CLASSES

Flexible Checker is not just a scoreboard but a combination of a scoreboard and the corresponding monitor feeding it. The monitors could be either monitors for protocols generated in house or from a third party. These monitors send transaction of the container transaction item type via the analysis ports. The idea behind instantiating monitors alongside the scoreboard is to bypass the process of instantiating an agent in passive mode and to add it to the env instead. This makes it simple to instantiate the monitor and set the corresponding configuration database in one place. Also, this gives the flexibility to change the number and type of instantiations. Reference code is shown below to support our in-house and third-party VIPs.

inhouse_monitor.sv

```
`ifndef __INHOUSE_MONITOR__
 `define __INHOUSE_MONITOR__

class inhouse_monitor extends flexible_monitor;
   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   task run;
      flexible_seq_item base_txn;
      inhousevip_seq_item txn;

      forever begin
         @(cb.clk);

         // create a transaction and assign the signals
         txn = inhousevip_seq_item::type_id::create("txn");
         txn.addr = cb.addr;
         txn.data = cb.data;

         // create the flexible transactions
         base_txn = txn;
         analysis_port.write(base_txn);
      end
   endtask: run
endclass: inhouse_monitor
`endif // __INHOUSE_MONITOR__

```

3rdPartyVIP_Monitor.sv

```
`ifndef __3rdparty_monitor__
`define __3rdparty_monitor__
class 3rdparty_monitor extends flexible_monitor;
   // This port receives the expvip_seq_item and puts it
   // in the container class derived from flexible_seq_item
   uvm_analysis_import #(extvip_seq_item) imp_port;
   extvip_monitor monitor;

   function new(string  name, uvm_component parent);
      imp_port = new("imp_port", this);
      monitor = extvip_monitor::type_id::create("monitor", this);
   endfunction: new

   function write(extvip_seq_item txn);
      3rdpartyvip_seq_item cntr_txn; // extended from flexible_seq_item
      flexible_seq_item send_txn;

      cntr_txn = 3rdparty_seq_item::type_id::create("cntr_txn");
      cntr_txn.txn_handle = txn;
      send_txn = cntr_txn;

      analysis_port.write(send_txn);
   endfunction: write

   function connect_phase(uvm_phase phase);
      monitor.analysis_port.connect(this.imp_port);
   endfunction: connect_phase

endclass: 3rdparty_monitor
`endif // __3rdparty_monitor__

```

VI. ELASTIC-/-FLEXIBLE SCOREBOARD

The crux of the scoreboard is its ability to add expected queues for each of the end nodes or targets. The idea is that whenever a transaction is reported at a source, it has inherent knowledge of which destination it must go to, or this information is provided to the scoreboard beforehand via a config item, then it pushes the transaction into the corresponding expected queue. Also, the transactions received by the scoreboard are casted to a base class type called the proxy transaction class, which enables the scoreboard to operate on a single transaction type. Once a transaction is reported by the target, it compares the transaction with the one in its corresponding expected queue. At the end of the simulation, the queues are checked to be empty, or the test fails and reports the stray transactions.

Flexible_Scoreboard.sv

```
`ifndef __FLEXIBLE_SCOREBOARD__
 `define __FLEXIBLE_SCOREBOARD__

`uvm_analysis_imp_decl(_m0_in_0)
`uvm_analysis_imp_decl(_m0_in_1)
`uvm_analysis_imp_decl(_m1_in_0)
`uvm_analysis_imp_decl(_m1_in_1)
`uvm_analysis_imp_decl(_m0_out_0)
`uvm_analysis_imp_decl(_m0_out_1)
`uvm_analysis_imp_decl(_m1_out_0)
`uvm_analysis_imp_decl(_m1_out_1)

// typedef for queues
typedef flexible_seq_item flex_item_queue[$];

virtual class flexible_scoreboard extends uvm_scoreboard;
   `uvm_component_utils(flexible_scoreboard)

   flexible_item_queue m0_array_of_queues_out[];
   flexible_item_queue m1_array_of_queues_out[];

   uvm_analysis_imp_m0_in_0 #(flexible_seq_item, flexible_scoreboard) m0_imp_in_0;
   uvm_analysis_imp_m0_in_1 #(flexible_seq_item, flexible_scoreboard) m0_imp_in_1;
   uvm_analysis_imp_m1_in_0 #(flexible_seq_item, flexible_scoreboard) m1_imp_in_0;
   uvm_analysis_imp_m1_in_1 #(flexible_seq_item, flexible_scoreboard) m1_imp_in_1;

   uvm_analysis_imp_m0_out_0 #(flexible_seq_item, flexible_scoreboard) m0_imp_out_0;
   uvm_analysis_imp_m0_out_0 #(flexible_seq_item, flexible_scoreboard) m0_imp_out_1;
   uvm_analysis_imp_m1_out_0 #(flexible_seq_item, flexible_scoreboard) m1_imp_out_0;
   uvm_analysis_imp_m1_out_0 #(flexible_seq_item, flexible_scoreboard) m1_imp_out_1;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction: new

   function build_phase(uvm_phase phase);
      m0_imp_in_0 = new("m0_imp_in_0", this);
      m0_imp_in_1 = new("m0_imp_in_1", this);
      m1_imp_in_0 = new("m1_imp_in_0", this);
      m1_imp_in_1 = new("m1_imp_in_1", this);
      m0_imp_out_0 = new("m0_imp_out_0", this);
      m0_imp_out_1 = new("m0_imp_out_1", this);
      m1_imp_out_0 = new("m1_imp_out_0", this);
      m1_imp_out_1 = new("m1_imp_out_1", this);

      m0_array_of_queues_out = new[2];
      m1_array_of_queues_out = new[2];

   endfunction: build_phase

   virtual function void check_phase(uvm_phase phase);
      super.check_phase(phase);

      foreach(m0_array_of_queues_out[i]) begin
         if(m0_array_of_queues_out[i].size()) begin
```

Flexible_Scoreboard.sv

```
      end
      foreach(m1_array_of_queues_out[i]) begin
         if(m1_array_of_queues_out[i].size()) begin
            `uvm_error(get_name(), $psprintf("m1_array_of_queues_out[%0d] is not empty [size:%0d]",
                                             i, m1_array_of_queues_out[i].size()))
         end
      end
   endfunction: check_phase

   virtual function void write_m0_in_0(flexible_seq_item txn);
      this.register_m0_input(txn);
   endfunction: write_m0_in_0

   virtual function void write_m0_in_1(flexible_seq_item txn);
      this.register_m0_input(txn);
   endfunction: write_m0_in_1

   virtual function void write_m1_in_0(flexible_seq_item txn);
      this.register_m1_input(txn);
   endfunction: write_m1_in_0

   virtual function void write_m1_in_1(flexible_seq_item txn);
      this.register_m1_input(txn);
   endfunction: write_m1_in_1

   virtual local function void register_m0_input(flexible_seq_item txn);
      m0_array_of_queues_out[txn.get_dest()].push_back(txn);
   endfunction: register_m0_input

   virtual local function void register_m1_input(flexible_seq_item txn);
      m1_array_of_queues_out[txn.get_dest()].push_back(txn);
   endfunction: register_m1_input

   virtual function void write_m0_out_0(flexible_seq_item txn);
      this.check_m0_output(txn);
   endfunction: write_m0_out_0

   virtual function void write_m0_out_1(flexible_seq_item txn);
      this.check_m0_output(txn);
   endfunction: write_m0_out_1

   virtual function void write_m1_out_0(flexible_seq_item txn);
      this.check_m1_output(txn);
   endfunction: write_m1_out_0

   virtual function void write_m1_out_1(flexible_seq_item txn);
      this.check_m1_output(txn);
   endfunction: write_m1_out_1

   virtual local function void check_m0_output(flexible_seq_item txn);
      flexible_seq_item exp_txn;
      exp_txn = m0_array_of_queues_out[txn.get_dest()].pop_front();
      txn.my_compare(exp_txn);
   endfunction: check_m0_output
```

VII. FLEXIBLE CHECKER

Flexible Checker gives the capability to check a transaction flowing through different nodes in a SoC. When a transaction is reported at a source, the checker populates the expected queues of all the internal nodes the transactions is expected to go to. Once a transaction is reported at these internal nodes, they are checked across the elements in their corresponding expected queues (which were populated by the source). While flowing through nodes, the transaction translates to different types/protocols, which requires coders to write a translation function that can then be reused across the scoreboard when such a translation is expected.

Flexible_Checker.sv

```
`ifndef __FLEXIBLE_CHECKER__
 `define __FLEXIBLE_CHECKER__

virtual class flexible_checker extends uvm_component;
   `uvm_component_utils(flexible_checker)

   // derived from the datastructure defined in our database of topology
   inhouse_monitor m0_monitors_in0;
   inhouse_monitor m0_monitors_in1;
   3rdparty_monitor m1_monitors_in0;
   3rdparty_monitor m1_monitors_in1;

   inhouse_monitor m0_monitors_out0;
   inhouse_monitor m0_monitors_out1;
   3rdparty_monitor m1_monitors_out0;
   3rdparty_monitor m1_monitors_out1;

   flexible_scoreboard m_scoreboard;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction: new

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      // building of required monitors
      m0_monitors_in0 = inhouse_monitor::type_id::create("m0_monitors_in0", this);
      m0_monitors_in1 = inhouse_monitor::type_id::create("m0_monitors_in1", this);
      m1_monitors_in0 = 3rdparty_monitor::type_id::create("m1_monitors_in0", this);
      m1_monitors_in1 = 3rdparty_monitor::type_id::create("m1_monitors_in1", this);

      m0_monitors_out0 = inhouse_monitor::type_id::create("m0_monitors_out0", this);
      m0_monitors_out1 = inhouse_monitor::type_id::create("m0_monitors_out1", this);
      m1_monitors_out0 = 3rdparty_monitor::type_id::create("m1_monitors_out0", this);
      m1_monitors_out1 = 3rdparty_monitor::type_id::create("m1_monitors_out1", this);
   endfunction: build_phase

   virtual function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);

      // connection of monitors to the respective ports in
      // scoreboard
      m0_monitors_in0.analysis_port.connect(m_scoreboard.m0_imp_in_0);
      m0_monitors_in1.analysis_port.connect(m_scoreboard.m0_imp_in_1);
      m1_monitors_in0.analysis_port.connect(m_scoreboard.m1_imp_in_0);
      m1_monitors_in1.analysis_port.connect(m_scoreboard.m1_imp_in_1);

      m0_monitors_out0.analysis_port.connect(m_scoreboard.m0_imp_out_0);
      m0_monitors_out1.analysis_port.connect(m_scoreboard.m0_imp_out_1);
      m1_monitors_out0.analysis_port.connect(m_scoreboard.m1_imp_out_0);
      m1_monitors_out1.analysis_port.connect(m_scoreboard.m1_imp_out_1);
   endfunction: connect_phase
endclass: flexible_checker

```

VIII. TRANSACTION TRACING

All transaction tracing is done using Flexible Checker, which includes

1. end-to-end checking to make transactions reach the correct port, in the correct order, and with the correct content,
2. selectively monitor any/all interfaces of an interconnect to trace the transaction route (or path) for correctness and flag an error otherwise,
3. selectively trace any/all transactions (using a preconfigured transaction ID) across an interconnect,

4. track timeout transactions (transactions that took more time than maximum allocated time in cycles),
5. track dropped transactions,
6. track split and merged transactions, and
7. track multicast and broadcast transactions.

IX. Statistic Monitoring & Dumping

Flexible Checker can also generate statistics, including latency, bandwidth, traffic distribution, and histogram data, with the transactions reported at the source and the destination and the relationship between them established by the scoreboard. The corresponding timing and delay information is stored on a master/ slave basis in the scoreboard and is then used to create textual or graphical data.

X. Conclusion

Flexible Checker was used in a highly configurable test bench and was able to do the checker functions just by passing the configuration parameters. It made building a block-level test bench redundant by using the transaction trace-checking feature, which helped the designer greatly in debugging failures as they were able to see the transaction flow and translation at each node of the design before reaching the target. Additionally, a statistics generator was used to print a table listing the transaction count, latency, and bandwidth numbers. This gave firsthand statistic numbers to the architects and gave a comparison point with the actual performance model numbers.